Apple Assembly Line

$1.80

Volume 7 -- Issue 3 December, 1986

In This Issue...

New ProDOS Book

Dennis Doms and Tom Weishaar, Technical Consultant and Publisher of Open-Apple, have conspired to bring us an interesting new book on programming under ProDOS, especially focussing on BASIC.SYSTEM.

"ProDOS Inside and Out" begins by explaining what an operating system is, progresses by describing files and directories, and goes on into simple commands. The next sections cover Applesoft programming and text file handling, followed by information about using machine language under BASIC.SYSTEM and using the ProDOS Kernel and MLI calls from BASIC.

This book does an excellent job of introducing the basic concepts of ProDOS, and then takes the reader on into quite advanced territory. It's very refreshing to find a book that doesn't assume you're already an expert and still has enough substance to help make you into one.

"ProDOS Inside and Out", by Dennis Doms and Tom Weishaar, from TAB Books. List is $16.95, we'll have it for $16 + shipping.

Fight Ways to Count Bits in a Byte........Bob Sander-Cederlof

Of course, there are always eight bits in a byte, by definition. But sometimes we want to know how many of those eight bits are 1's. There are enough reasons to generate this count that some computers have a special machine language opcode to count the number of 1-bits in a byte or word.

One reason that comes to mind is to compute the odd- or even-parity bit for a byte of ASCII data. Another is in processing of picture data in a computer vision system.

I came up with at least eight different programs to count the 1-bits in a byte. I would choose one based on how critical speed or memory-usage is in a particular case.

The fastest one, table lookup, can translate a byte to a bit count in 4 cycles (7 if you also count getting the byte-value into X or Y. It is fast, but it requires a 256-byte table. The table has the counts for each possible value. If speed is critical, I would use this method. I would probably use one of the other methods to create the table during initialization, rather than assembling it from source code. An example of the table lookup method is shown in lines 2130-2160, as a subroutine.

The next method that I thought of is shown in lines 1500-1590. I count the 1-bits in the X-register, while shifting the data byte. The loop runs eight times, once for each bit position. This one takes from 86 cycles to 94 cycles, depending on how many 1-bits there are. The average time is 90 cycles. (I am not counting the JSR and RTS.)

A slightly faster method checks two bit positions during each loop (lines 1610-1730). The reduction in loop overhead changes the times to minimun 66, maximum 74, average 70 cycles.

The fastest method I found without table lookup is shown in lines 1750-2010. This one is completely unwrapped, so there is no loop overhead at all. The times come out to minimum 43, maximum 51, average 47 cycles.

I tried just optimizing the first method, and lines 2030-2110 are the result. This one gets more average speed because it loops only until there are no more 1-bits left. The minimum is only 8 cycles (when the byte = $00). If the byte = $80, it takes 14 cycles. For $C0 or $40, it takes 128 or 120 cycles. For values that have bit 0 = 1, it will take from 70 to 77 cycles, depending on the number of 1-bits in the other seven bit positions. The overall average is 65 cycles.

By changing this last method just a little, the overall average can be reduced to between 58 and 59 cycles. The result is still fairly small, so I think this one would be may favorite choice when the blinding speed of a table lookup is not necessary:

S-C Macro Assembler Version 2.0......DOS $100, ProDOS $100, both for $120
Version 2.0 DOS Upgrade Kit for 1.0/1.1/1.2 owners....................$20
ProDOS Upgrade Kit for Version 2.0 DOS owners.........................$30
Source Code of S-C Macro 2.0 (DOS only)..................additional $100
Full Screen Editor for S-C Macro (with complete source code)..........$49
S-C Cross Reference Utility......without source code $20, with source $50
RAK-Ware DISASM..................without source code $30, with source $50
S-C Word Processor (with complete source code)........................$50
DP18 Source and Object..$50
Double Precision Floating Point for Applesoft (with source code)......$50
ES-CAPE (Extended S-C Applesoft Program Editor)..........
Including Version 2.0 With Source Code..........$50
ES-CAPE Version 2.0 and Source Code Update (for Registered Owners)....$30
Bag of Tricks 2 (Quality Software)........................($49.95) $45 *
MacASM -- Macro Assembler for MacIntosh (Mainstay).........($150.00) $100 *
S-C Documentor (complete commented source code of Applesoft ROMs).....$50
Cross Assemblers for owners of S-C Macro Assembler.....$32.50 to $50 each
(Available: 6800/1/2, 6301, 6805, 6809, 68000, Z-80, Z-8, 8048,
8051, 8085, 1802/4/5, PDP-11, GI1650/70, others)

AAL Quarterly Disks.........................each $15, or any four for $45
Each disk contains the source code from three issues of AAL:
Jan-Mar, Apr-Jun, Jul-Sep, and Oct-Dec.
(All source code is formatted for S-C Macro Assembler. Other assemblers require some effort to convert file type and edit directives.)

Diskettes (with hub rings)........................ package of 25 for $15 *
Vinyl disk pages, 6"x8.5", hold two disks each.................10 for $6 *
Diskette Mailing Protectors (hold 1 or 2 disks).............40 cents each
(Cardboard folders designed to fit 6"X9" Envelopes.) or $25 per 100 *
Envelopes for Diskette Mailers............................ 6 cents each

65802 Microprocessor (Western Design Center)..................($95) $50 *
quikLoader EPROM System (SCRG)...............................($179) $170 *
PROmGRAMER (SCRG)..($149.50) $140 *
Switch-a-Slot (SCRG).......................................($179.50) $170 *
Minuteman 250 Uninterruptible Power Supply...................($359) $320 +

"65816/65802 Assembly Language Programming", Fischer........($19.95) $18 *
"Programming the 65816", Eyes...............................($22.95) $21 *
"Apple //e Reference Manual", Apple Computer................($24.95) $23 *
"Apple //c Reference Manual", Apple Computer................($24.95) $23 *
"ProDOS Technical Reference Manual", Apple Computer.........($29.95) $27 *
"Apple ProDOS: Advanced Features for Programmers", Little..($17.95) $17 *
"ProDOS Inside and Out", Doms & Weishaar....................($16.95) $16 *
"Beneath Apple ProDOS", Worth & Lechner.....................($19.95) $18 *
"Beneath Apple DOS", Worth & Lechner........................($19.95) $18 *
"Inside the Apple //c", Little..............................($19.95) $18 *
"Inside the Apple //e", Little..............................($19.95) $18 *
"Understanding the Apple //e", Sather.......................($24.95) $23 *
"Understanding the Apple II", Sather........................($22.95) $21 *
"Apple II+/IIe Troubleshooting & Repair Guide", Brenner.....($19.95) $18 *
"Apple][Circuit Description", Gayler......................($22.95) $21 *
"Assem. Language for Applesoft Programmers", Finley & Myers.($18.95) $18 *
"Enhancing Your Apple II, vol. 1", Lancaster................($15.95) $15 *
"Enhancing Your Apple II, vol 2", Lancaster.................($17.95) $17 *
"Assembly Cookbook for the Apple II/IIe", Lancaster.........($21.95) $20 *
"Real Time Programming -- Neglected Topics", Foster..........($9.95) $9 *
"Microcomputer Graphics", Myers.............................($14.95) $14 *
"Assembly Lines -- the Book", Wagner........................($19.95) $18 *
"AppleVisions", Bishop & Grossberger........................($39.95) $36 *

* On these items add $2.00 for the first item and
$.75 for each additional item for US shipping.
+ Inquire for shipping cost.
Foreign customers inquire for postage needed.
Texas residents please add 6 1/8 % sales tax to all orders.

*** S-C SOFTWARE, P. O. BOX 280300, Dallas, TX 75228 ***
*** (214) 324-2050 ***
*** Master Card, VISA, Discover and American Express ***

```
COUNT   LDX #0
        LDA BYTE
        BEQ .3          value is 00000000
        BPL .2          bit 7 = 0
.1      INX             count the 1-bit
.2      ASL             shift the value
        BMI .1          ...its 1, count it
        BNE .2          ...its 0, do not count
.3      RTS             finished
```

Lines 2270-2460 implement still another method, which always takes 57 cycles regardless of the mix of zeroes and ones. It uses an extra pagezero location for a temporary cell. The TAX in line 2450 is required only if it is really required that the bit count be returned in the X-register. My test program has this requirement, but there is no reason to force that requirement on a real program. If the A-register is good enough, then this method only takes 55 cycles.

Lines 2180-2250 are "half" a table lookup. That is, I am using half a table: entries for values from $00 to $7F. This is a tiny bit slower than the full table lookup, but saves 128 bytes of table. I shift the value right one bit position, guaranteeing a value from $00 to $7F, and save the bit shifted out in carry. Then adding carry to the value from the table gives me the count for the whole original value. A slight change to how carry is added can reduce the average time by half a cycle:

```
COUNT   LDA BYTE
        LSR
        TAY
        LDX TBL1,Y
        BCC .1
        INX
.1      RTS
```

The wildest way I came up with is based on one I read about in the latest issue of Byte magazine (the IBM special issue, in the BIX section). By masking, adding, and shifting, the bits can be all aligned and added into a count. Lines 2480-2720 are my code for this method. The time is 58 cycles, regardless of the 0/1 mix.

I wrote a test routine, so I could tell whether my methods really worked or not. Lines 1070-1480 call each of three different bit-counters for every possible value of the byte. It keeps calling the first method until I hit the space bar, and then advances to the second method. Then it keeps calling the second, until another key-tap advances it to the third method. I can keep cycling through the methods this way, until I type a RETURN to end it. Inside each of the three loops I have a "LDA $C030" instruction, to toggle the speaker. The three loops are identical in timing except for the bit-counting code itself. The result is that I can tell by "ear" which methods are fastest, and which ones have a constant time regardless of the 0/1 mix. I tested the accuracy by comparing the results in TBL1 and TBL2 with the monitor "V" command, and by displaying the TBL using the monitor.

```
                 *SAVE S.BIT COUNTERS
                 *--------------------------------
00-              BYTE     .EQ $00
01-              B        .EQ $01
01-              SUM1     .EQ $01
02-              SUM2     .EQ $02
                 *--------------------------------
                 T
0800- A9 00               LDA #0
0802- 85 00               STA BYTE
                 *--------------------------------
0804- 20 56 08   .1       JSR COUNT.1
0807- A4 00               LDY BYTE
0809- 8A                  TXA
080A- 99 00 0A            STA TBL1,Y
080D- AD 30 C0            LDA $C030
0810- E6 00               INC BYTE
0812- AD 00 C0            LDA $C000
0815- 10 ED               BPL .1
0817- 8D 10 C0            STA $C010
081A- C9 8D               CMP #$8D
081C- F0 37               BEQ .99
                 *--------------------------------
081E- 20 64 08   .2       JSR COUNT.2
0821- A4 00               LDY BYTE
0823- 8A                  TXA
0824- 99 00 0B            STA TBL2,Y
0827- AD 30 C0            LDA $C030
082A- E6 00               INC BYTE
082C- AD 00 C0            LDA $C000
082F- 10 ED               BPL .2
0831- 8D 10 C0            STA $C010
0834- C9 8D               CMP #$8D
0836- F0 1D               BEQ .99
                 *--------------------------------
0838- 20 76 08   .3       JSR COUNT.3
083B- A4 00               LDY BYTE
083D- 8A                  TXA
083E- 99 00 0B            STA TBL2,Y
0841- AD 30 C0            LDA $C030
0844- E6 00               INC BYTE
0846- AD 00 C0            LDA $C000
0849- 10 ED               BPL .3
084B- 8D 10 C0            STA $C010
084E- C9 8D               CMP #$8D
0850- F0 03               BEQ .99
                 *--------------------------------
0852- 4C 04 08            JMP .1
0855- 60         .99      RTS
                 *--------------------------------
                 COUNT.1
0856- A0 08               LDY #8         TEST 1 BIT AT A TIME,
0858- A2 00               LDX #0            LOOPING 8 TIMES
085A- A5 00               LDA BYTE
085C- 0A         .1       ASL            IF BIT = 1, COUNT IT
085D- 90 01               BCC .2
085F- E8                  INX
0860- 88         .2       DEY
0861- D0 F9               BNE .1         NEXT BIT
0863- 60                  RTS
                 *--------------------------------
                 COUNT.2
0864- A0 04               LDY #4         TEST 2 BITS AT A TIME,
0866- A2 00               LDX #0            LOOPING 4 TIMES
0868- A5 00               LDA BYTE
086A- 0A         .1       ASL
086B- 10 01               BPL .2         IF BIT = 1, COUNT IT
086D- E8                  INX
086E- 90 01      .2       BCC .3         IF BIT = 1, COUNT IT
0870- E8                  INX
0871- 0A         .3       ASL
0872- 88                  DEY
0873- D0 F5               BNE .1         NEXT PAIR OF BITS
0875- 60                  RTS
```


Personalize and
Protect your Apple
with the
Banana Chip©
When you turn on your Apple™, this remarkable new ROM replacement will display your name or personalized message with optional password protection.
master charge
THE INTERBANK CARD
VISA
$39.95
CALL TOLL FREE:
1-800-843-0975
Nationwide
1-800-443-9060
In California
Azure Mountain
Products
14755 Ventura Blvd., Ste. 1483 Sherman Oaks, CA 91403

```
                        *-------------------------------
                        COUNT.3
0876- A2 00                        LDX #0        NO LOOPS, JUST STRAIGHT-LINE CODE
0878- A5 00                        LDA BYTE
087A- 10 01                        BPL .1        BIT 7
087C- E8                           INX
087D- 4A                .1         LSR
087E- 90 01                        BCC .2        BIT 0
0880- E8                           INX
0881- 4A                .2         LSR
0882- 90 01                        BCC .3        BIT 1
0884- E8                           INX
0885- 4A                .3         LSR
0886- 90 01                        BCC .4        BIT 2
0888- E8                           INX
0889- 4A                .4         LSR
088A- 90 01                        BCC .5        BIT 3
088C- E8                           INX
088D- 4A                .5         LSR
088E- 90 01                        BCC .6        BIT 4
0890- E8                           INX
0891- 4A                .6         LSR
0892- 90 01                        BCC .7        BIT 5
0894- E8                           INX
0895- 4A                .7         LSR
0896- 90 01                        BCC .8        BIT 6
0898- E8                           INX
0899- 60                .8         RTS
                        *-------------------------------
                        COUNT.4
089A- A2 00                        LDX #0
089C- A5 00                        LDA BYTE
089E- F0 06                        BEQ .3
08A0- 10 01             .1         BPL .2
08A2- E8                           INX
08A3- 0A                .2         ASL
08A4- D0 FA                        BNE .1
08A6- 60                .3         RTS
                        *-------------------------------
                        COUNT.5
08A7- A4 00                        LDY BYTE
08A9- BE 00 0A                     LDX TBL1,Y
08AC- 60                           RTS
                        *-------------------------------
                        COUNT.6
08AD- A5 00                        LDA BYTE
08AF- 4A                           LSR
08B0- AA                           TAX
08B1- A9 00                        LDA #0
08B3- 7D 00 0A                     ADC TBL1,X
08B6- AA                           TAX
08B7- 60                           RTS
                        *-------------------------------
                        COUNT.7
08B8- A5 00                        LDA BYTE                3
08BA- 4A                           LSR           BIT 0     2
08BB- 85 01                        STA B                   3
08BD- A9 00                        LDA #0                  2
08BF- 2A                           ROL                     2
08C0- 46 01                        LSR B         BIT 1     5
08C2- 69 00                        ADC #0
08C4- 46 01                        LSR B         BIT 2
08C6- 69 00                        ADC #0
08C8- 46 01                        LSR B         BIT 3
08CA- 69 00                        ADC #0
08CC- 46 01                        LSR B         BIT 4
08CE- 69 00                        ADC #0
08D0- 46 01                        LSR B         BIT 5
08D2- 69 00                        ADC #0
08D4- 46 01                        LSR B         BIT 6
08D6- 65 01                        ADC B         BIT 7
08D8- AA                           TAX
08D9- 60                           RTS
                        *-------------------------------
```

```
                 2480 COUNT.8
08DA- A5 00      2490        LDA BYTE
08DC- 29 55      2500        AND #$55      BITS 6,4,2,0
08DE- 85 01      2510        STA SUM1
08E0- 45 00      2520        EOR BYTE      BITS 7,5,3,1
08E2- 4A         2530        LSR           CLEARS CARRY
08E3- 65 01      2540        ADC SUM1      FORM wwxxyyzz, where each pair
08E5- 85 01      2550        STA SUM1           is 0, 1, or 2.
08E7- 29 33      2560        AND #$33      Isolate 00xx00zz
08E9- 85 02      2570        STA SUM2
08EB- 45 01      2580        EOR SUM1      Isolate ww00yy00
08ED- 4A         2590        LSR
08EE- 4A         2600        LSR           Now it is 00ww00yy
08EF- 65 02      2610        ADC SUM2      Form 0uuu0vvv, where each group
08F1- 85 02      2620        STA SUM2           is 0, 1. 2. 3. or 4.
08F3- 29 0F      2630        AND #$0F      Isolate 00000vvv
08F5- 85 01      2640        STA SUM1
08F7- 45 02      2650        EOR SUM2      Isolate 0uuu0000
08F9- 4A         2660        LSR
08FA- 4A         2670        LSR
08FB- 4A         2680        LSR
08FC- 4A         2690        LSR           Now it is 00000uuu
08FD- 65 01      2700        ADC SUM1      Form count 0-8
08FF- AA         2710        TAX
0900- 60         2720        RTS
                 2730 *--------------------------------
0901-            2740        .BS *+255/256*256-*
                 2750 *--------------------------------
0A00-            2760 TBL1   .BS 256
0B00-            2770 TBL2   .BS 256
                 2780 *--------------------------------
```

Apple //gs ROM Checksummer................Bob Sander-Cederlof

Somewhere I ran across a statement that the 128K bytes of ROM in the //gs have a standard checksum. A value is stored at $FFFFF6 and $FFFFF7 to pad out the checksum, so that it will always add up to $1234. I tried various ways of adding up the bytes to get that value, and came up with the following little program.

```
                        1000 *SAVE S.CHECKSUM
                        1010 *--------------------------------
                        1020        .OP 65816
                        1030 *--------------------------------
                        1040 FORM.AND.DISPLAY.IIGS.CHECKSUM
000800- 18              1050        CLC              ENTER NATIVE MODE
000801- FB              1060        XCE
000802- C2 30           1070        REP #$30         FULL 16
                        1080 *--------------------------------
000804- A2 00 00        1090        LDX ##0          0 TO FFFF
000807- 8A              1100        TXA              START WITH A=0
000808- 38              1110        SEC              WHY?  IT MAKES ANSWER $1234
000809- 7F 00 00 FE     1120 .1     ADC $FE0000,X        BANK $FE
00080D- E8              1130        INX
00080E- E8              1140        INX                  TWO BYTES AT A TIME
00080F- D0 F8           1150        BNE .1               ...UNTIL WHOLE BANK
000811- 7F 00 00 FF     1160 .2     ADC $FF0000,X        BANK $FF
000815- E8              1170        INX
000816- E8              1180        INX                  TWO BYTES AT A TIME
000817- D0 F8           1190        BNE .2               ...UNTIL WHOLE BANK
000819- 85 00           1200        STA 0                SAVE RESULT AT $00.01
                        1210 *--------------------------------
00081B- 38              1220        SEC              EMULATION MODE
00081C- FB              1230        XCE
00081D- A5 01           1240        LDA 1            PRINT CHECKSUM WITH OLD MONITOR
00081F- 20 DA FD        1250        JSR $FDDA            SUBROUTINE 'PRBYTE'
000822- A5 00           1260        LDA 0
000824- 20 DA FD        1270        JSR $FDDA
000827- 60              1280        RTS
                        1290 *--------------------------------
```

The 128K ROM occupies the space from $FE0000 through $FFFFFF. My program adds up the data there two bytes at a time in 16-bit registers. Doing a normal add of these "words" gave a sum of $1233, so I started with SEC instead of CLC to get a sum of $1234. I took my program to a computer store and tried it on a "real" //gs, with a different ROM set, and got the same result: $1234.

If you make any changes to the ROMs yourself, be sure to fix the checksum too. Otherwise you may not even be able to boot!

Who Worked on the //gs?..................Bob Sander-Cederlof

You can see a list of the names of the people inside Apple who worked on the //gs by typing in the following program. The names are stored inside the //gs ROM, organized according to which project they worked on. I suppose future versions of the //gs ROMs may not have this information (they may need the space for more useful tools), or it may be moved around, but so far it is all the machines I have looked at.

The names start at $BA0B, and are in two consecutive blocks terminated by a 00 byte. Most of the bytes are ASCII characters with bit 7 high (=1). Whenever you find a byte with bit 7 low (=0), it is a repeat count for the following character. Thus 07 A0 means repeat $A0 seven times, or print seven blanks. You will also find repeat counts followed by $53, which is an inverse S. However, in MouseText, it is a horizontal line. Evidently it is supposed to be displayed with MouseText turned on.

There may be a program somewhere inside the ROM that prints the list of names, but I haven't even tried finding it yet. Anyway, the following one will do it.

```
                        1000 *SAVE S.NAMES.IIGS
                        1010 *--------------------------------
                        1020 *      PRINT NAMES OF APPLE //GS DEVELOPERS
                        1030 *--------------------------------
                        1040        .OP 65816
                        1050 *--------------------------------
                        1060 PRINT.NAMES.OF.APPLE.PEOPLE
000800- 18              1080        CLC              ENTER NATIVE MODE
000801- FB              1090        XCE
000802- C2 10           1100        REP #$10         X,Y 16-BIT MODE
000804- A9 8D           1101        LDA #$8D         SKIP TWO LINES
000806- 20 3B 08        1102        JSR MY.COUT
000809- 20 3B 08        1103        JSR MY.COUT
00080C- A2 0A BA        1110        LDX ##$BA0A      NAMES START AT $BA0B
00080F- 20 18 08        1120        JSR PRINT.NAMES     FIRST BLOCK
000812- 20 18 08        1130        JSR PRINT.NAMES     SECOND BLOCK
000815- 38              1140        SEC
000816- FB              1150        XCE              BACK TO NATIVE MODE
000817- 60              1190        RTS
                        1200 *--------------------------------
                        1210 PRINT.NAMES
000818- E8              1220 .1     INX              NEXT CHAR
000819- BF 00 00 FF     1230        LDA $FF0000,X
00081D- F0 07           1240        BEQ .2           ...END OF A BLOCK
00081F- 10 06           1250        BPL .3           ...REPEAT COUNT
000821- 20 3B 08        1260        JSR MY.COUT      PRINT THE CHAR
000824- 80 F2           1270        BRA .1
000826- 60              1280 .2     RTS              RETURN
                        1290 *--------------------------------
000827- A8              1320 .3     TAY              REPEAT COUNT TO Y-REG
000828- E8              1340        INX
000829- BF 00 00 FF     1350        LDA $FF0000,X   GET REPEATED CHAR
00082D- C9 53           1351        CMP #$53         MOUSE TEXT LINE?
00082F- D0 02           1352        BNE .4           ...NO
000831- A9 AD           1353        LDA #"-"         ...YES, SUBSTITUTE DASH
000833- 20 3B 08        1360 .4     JSR MY.COUT      PRINT THE CHAR
000836- 88              1370        DEY              N TIMES
000837- D0 FA           1380        BNE .4
000839- 80 DD           1390        BRA .1
                        1400 *--------------------------------
                        1410 MY.COUT
00083B- 48              1420        PHA              SAVE EVERYTHING!!!
00083C- DA              1430        PHX
00083D- 5A              1440        PHY
00083E- 08              1450        PHP
00083F- 38              1460        SEC              EMULATION MODE
000840- FB              1470        XCE
000841- 08              1480        PHP
000842- 20 ED FD        1490        JSR $FDED
000845- 28              1500        PLP              ORIGINAL MODE
000846- FB              1510        XCE
000847- 28              1520        PLP              RESTORE REGISTERS
000848- 7A              1530        PLY
000849- FA              1540        PLX
00084A- 68              1550        PLA
00084B- 60              1560        RTS
                        1570 *--------------------------------
```

Commented Listing of ProDOS, $DE00-DEF2.....Bob Sander-Cederlof

What happens when you call ProDOS MLI? In assembly language, MLI calls look like this:

```
       JSR $BF00
       .DA #command,IOB.Address
```

The instruction at $BF00 is a "JMP $BFB7" in ProDOS 1.1.1; it is possibly different in other versions. All of the following disassembly is for ProDOS 1.1.1. The changes in the new ProDOS 1.2 are minor, and if you have 1.2 you should be able to figure out what they are.

At $BFB7 there is some code I call LC.BRIDGE.ENTRY. It "remembers" what language card areas are switched in at $D000 and at $E000, and then turns on the language card so that it can jump into the MLI call processor.

```
       BFB7:  SEC      Set flag
              ROR MLI.ACTIVE.FLAG
              LDA $E000
              STA E000.BYTE  (BFF4)
              LDA $D000
              STA D000.BYTE  (BFF5)
              LDA $C08B
              LDA $C08B
              JMP $DE00
```

Now comes the good part. The following listing is of the code starting at $DE00, which decodes the bytes following your JSR $BF00 and performs your request.

Lines 1010-1080 define some page-zero variables used by MLI. Lines 1090-1220 define some items in the system global page. Lines 1230-1280 define some entry points inside the rest of MLI, not listed here.

MLI calls don't change the X and Y registers, so they are saved at line 1390. The return address (of the JSR $BF00) is pulled off the stack and saved at PARM.PNTR in page zero, so that it can be used to access your command code and IOB address. Lines 1410-1490 also compute the address of the next instruction, to be used later for a return address. This address is saved in the system global page, and is useful sometimes for debugging. (We have published several articles on enhanced error messages and tracers for MLI calls in previous issues of AAL.)

Lines 1500-1650 convert the command code to an index by a strange scheme. The legal command codes are (in hex): 40, 41, 65, 80 thru 82, and C0 thru D3. The hashing algorithm used here adds the high nybble of the command code to the whole code, and then masks it to the lower five bits. This compresses the range of the codes, without any overlapping.

```
       40,41 --> 04,05        C0-CF --> 0C-1B
       65    --> 0B           D0-D2 --> 1D-1F
       80-82 --> 08-0A        D3    --> 00
```

This index is used then to look into the COMMAND.HASH.TABLE, which has the actual command codes in the indexed positions. If the original code is not found there, then the original code was an illegal command number. The hash index is also used to look up the parameter count in PARM.CNT.TABLE. I have appended the code for these two tables to the end of today's listing, at lines 3100 to the end.

Lines 1810-1920 branch various ways according to the command code. Most of the commands are not shown in this listing, but most of the code for READ BLOCK and WRITE BLOCK is shown (lines 2690-3080). When a command is finished, it eventually finds its way back to EXIT.TO.CALLER at line 2180.

Lines 2180-2560 get us back to our own code again, after the JSR $BF00. If the MLI call produced an error, the code number for that error will be in SYS.ERRNUM. The error code will be returned in the A-register, with carry SET. If there is no error to report, A=0 and carry is clear.

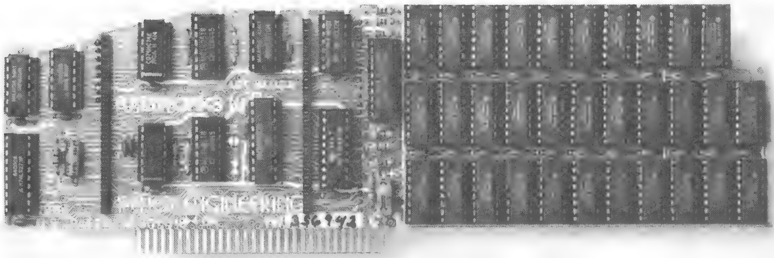
We will probably be presenting more sections of MLI disassembly in the near future. You may remember that we published portions of an earlier ProDOS version back in November and December of 1983.

```
                  1000 *SAVE S.MLI.DE00.DEF2
                  1010 *-------------------------------
40-               1020 PARM.PNTR   .EQ $40.41
42-               1030 COMMAND     .EQ $42
43-               1040 UNIT.NO     .EQ $43
44-               1050 BUFF.PNTR   .EQ $44.45
46-               1060 BLOCK.NO    .EQ $46.47
48-               1070 GEN.PNTR1   .EQ $48.49
4E-               1080 GEN.PNTR2   .EQ $4E,4F
                  1090 *-------------------------------
BF03-             1100 CALL.QUIT           .EQ $BF03
BF06-             1110 CALL.TIME           .EQ $BF06
BF09-             1120 CALL.SYSERR         .EQ $BF09
BF0F-             1130 SYS.ERRNUM          .EQ $BF0F
BF10-             1140 DRIVER.ADDR.TABLE   .EQ $BF10 thru BF2F
BF95-             1150 BACKUP.BIT          .EQ $BF95
BF9B-             1160 MLI.ACTIVE.FLAG     .EQ $BF9B
BF9C-             1170 MLI.RETURN          .EQ $BF9C.D
BF9E-             1180 MLI.X               .EQ $BF9E
BF9F-             1190 MLI.Y               .EQ $BF9F
BFA0-             1200 LC.BRIDGE.EXIT      .EQ $BFA0
BFF4-             1210 E000.BYTE           .EQ $BFF4
BFF5-             1220 D000.BYTE           .EQ $BFF5
                  1230 *-------------------------------
DEF3-             1240 INTERRUPT.HANDLER   .EQ $DEF3
E047-             1250 FILING.FUNCTIONS    .EQ $E047
FC9F-             1260 CHECK.IF.MEM.FREE   .EQ $FC9F
                  1270 *-------------------------------
FEF5-             1280 JUMP                .EQ $FEF5.6
                  1290 *-------------------------------
                  1300        .OR $DE00
                  1310        .TA $800
                  1320 *-------------------------------
                  1330 *   JSR $BF00 comes here
                  1340 *   .DA #$xx  command byte
                  1350 *   .DA xxxx  IOB Address
                  1360 *-------------------------------
                  1370 MLI.ENTRY
DE00- D8          1380        CLD
DE01- 8C 9F BF    1390        STY MLI.Y
DE04- 8E 9E BF    1400        STX MLI.X
DE07- 68          1410        PLA          GET RETURN ADDRESS
DE08- 85 40       1420        STA PARM.PNTR     WILL POINT AT BYTES
DE0A- 18          1430        CLC               FOLLOWING JSR $BF00
DE0B- 69 04       1440        ADC #4       COMPUTE ACTUAL RETURN
DE0D- 8D 9C BF    1450        STA MLI.RETURN    AND SAVE FOR LATER
```

```
DE10- 68                    PLA
DE11- 85 41                 STA PARM.PNTR+1
DE13- 69 00                 ADC #0
DE15- 8D 9D BF              STA MLI.RETURN+1
               *---Check Command Code-----------
DE18- A0 00                 LDY #0
DE1A- 8C 0F BF              STY SYS.ERRNUM
DE1D- C8                    INY
DE1E- B1 40                 LDA (PARM.PNTR),Y
DE20- 4A                    LSR          Hash it (CC/16 + CC) & $1F
DE21- 4A                    LSR
DE22- 4A                    LSR
DE23- 4A                    LSR
DE24- 18                    CLC
DE25- 71 40                 ADC (PARM.PNTR),Y
DE27- 29 1F                 AND #$1F
DE29- AA                    TAX          Use hashcode as index
DE2A- B1 40                 LDA (PARM.PNTR),Y  Original command code
DE2C- DD 65 FD              CMP COMMAND.HASH.TABLE,X
DE2F- D0 76                 BNE ERR.CALL.NO    Not valid command
               *---Get IOB Address--------------
DE31- C8                    INY
DE32- B1 40                 LDA (PARM.PNTR),Y
DE34- 48                    PHA
DE35- C8                    INY
DE36- B1 40                 LDA (PARM.PNTR),Y
DE38- 85 41                 STA PARM.PNTR+1
DE3A- 68                    PLA
DE3B- 85 40                 STA PARM.PNTR
               *---Check Parm Count-------------
DE3D- A0 00                 LDY #0
DE3F- BD 85 FD              LDA PARM.CNT.TABLE,X
DE42- F0 1C                 BEQ MLI.GETTIME   ...only one with 0 parms
DE44- D1 40                 CMP (PARM.PNTR),Y
DE46- D0 63                 BNE ERR.PARM.CNT
               *---Branch Various Ways----------
DE48- BD 65 FD              LDA COMMAND.HASH.TABLE,X
DE4B- C9 65                 CMP #$65
DE4D- F0 0E                 BEQ .1       ...QUIT CALL
DE4F- 0A                    ASL
DE50- 10 14                 BPL MLI.RWBLK        $80 or $81
DE52- B0 1D                 BCS MLI.CX.AND.DX    $Cx or $Dx
DE54- 4A                    LSR                  $40 or $41
DE55- 29 03                 AND #$03
DE57- 20 F3 DE              JSR INTERRUPT.HANDLER
DE5A- 4C 78 DE              JMP EXIT.TO.CALLER
DE5D- 4C 03 BF       .1     JMP CALL.QUIT        $65
               *--------------------------------
               *   Command $82. Get the Date and Time
               *--------------------------------
               MLI.GETTIME
DE60- 20 06 BF              JSR CALL.TIME
DE63- 4C 78 DE              JMP EXIT.TO.CALLER
               *--------------------------------
               *   Commands $80 and $81
               *--------------------------------
               MLI.RWBLK
DE66- 4A                    LSR          Make $00 and 01
DE67- 69 01                 ADC #1       Into $01 and 02
DE69- 85 42                 STA COMMAND  Store into command block
DE6B- 20 B2 DE              JSR BLOCK.IO.SETUP  Do the I/O
DE6E- 4C 78 DE              JMP EXIT.TO.CALLER
               *--------------------------------
               *   Commands $C0 thru $D3
               *--------------------------------
               MLI.CX.AND.DX
DE71- 4A                    LSR          Make command code into
DE72- 29 1F                 AND #$1F         an index
DE74- AA                    TAX
DE75- 20 47 E0              JSR FILING.FUNCTIONS
               *---fall into EXIT routine-------
               * (DE78) DE5A DE63 DE6E DEB0       callers
               EXIT.TO.CALLER
DE78- A9 00                 LDA #0       Clear BACKUP bit
DE7A- 8D 95 BF              STA BACKUP.BIT
DE7D- AC 0F BF              LDY SYS.ERRNUM   If any error code,
DE80- C0 01                 CPY #1           then set carry
DE82- 98                    TYA              and clear Z-bit
DE83- 08                    PHP              Save this status
```

RamWorks® III

Patented Performance from the Recognized Leader



With battery backed RAM port, RGB port, increased memory capacity, full software compatibility and more compact design, RamWorks III is a generation ahead.

RamWorks III is the newest 3rd generation RAM card for the Apple IIe. It incorporates all of the technology and improvements that years of experience and over a hundred thousand sales have given us. By selling more memory cards than anyone else and listening to our customers, we were able to design a memory card that has the ultimate in performance, quality, compatibility and ease of use. A design so advanced it's patented. We call it RamWorks III, you'll call it awesome!

The AppleWorks Amplifier.

While RamWorks III is recognized by all memory intensive programs, NO other expansion card comes close to offering the multitude of enhancements to AppleWorks that RamWorks III does. Naturally, you'd expect RamWorks III to expand the available desktop, after all Applied Engineering was a year ahead of everyone else *including Apple* in offering more than 55K, and we still provide the largest AppleWorks desktops available. But a larger desktop is just part of the story. Look at all the AppleWorks enhancements that even Apple's own card does not provide and *only* RamWorks III does. With a 256K or larger RamWorks III, *all* of AppleWorks (including printer routines) will automatically load itself into RAM dramatically increasing speed by eliminating the time required to access the program disk drive. Switch from word processing to spreadsheet to database at the speed of light with no wear on disk drives.

Only RamWorks eliminates AppleWorks' internal memory limits, increasing the maximum number of records available from 1,350 to over 25,000. *Only* RamWorks increases the number of lines permitted in the word processing mode from 2,250 to over 15,000. And *only* RamWorks offers a built-in printer buffer, so you won't have to wait for your printer to stop before returning to AppleWorks. RamWorks even expands the clipboard. And auto segments large files so they can be saved on two or more disks. You can even have Pinpoint or MacroWorks and your favorite spelling checker in RAM for instant response.

RamWorks, nothing comes close to enhancing AppleWorks so much.

The Most Friendly, Most Compatible Card Available.

Using RamWorks III couldn't be easier because it's compatible with more off-the-shelf software than any other RAM card. Popular programs like AppleWorks, Pinpoint, Catalyst, MouseDesk, HowardSoft, FlashCalc, Pro-Filer, Managing Your Money, SuperCalc 3a, and MagiCalc to name a few (and *all* hardware add on's like ProFile and Sider hard disks). RamWorks is even compatible with software written for Apple cards. But unlike other cards, RamWorks plugs into the IIe auxiliary slot providing our super sharp 80 column text (U.S. Patent #4601081) in a completely integrated system while leaving expansion slots 1 through 7 available for other peripheral cards.

RamWorks III is compatible with all Apple IIe's, enhanced, unenhanced, American or European versions.

Highest Memory Expansion.

Applied Engineering has always offered the largest memory for the IIe and RamWorks III continues that tradition by expanding to 1 full MEG on the main card using standard RAMs, more than most will ever need (1 meg is about 500 pages of text)...but if you do ever need more than 1 MEG, RamWorks III has the widest selection of expander cards available. Additional 512K, 2 MEG, or 16 MEG cards just snap directly onto RamWorks III by plugging into the industry's only low profile (no slot 1 interference) fully decoded memory expansion connector. You can also choose non-volatile, power independent expanders allowing permanent storage for up to 20 years.

It Even Corrects Mistakes.

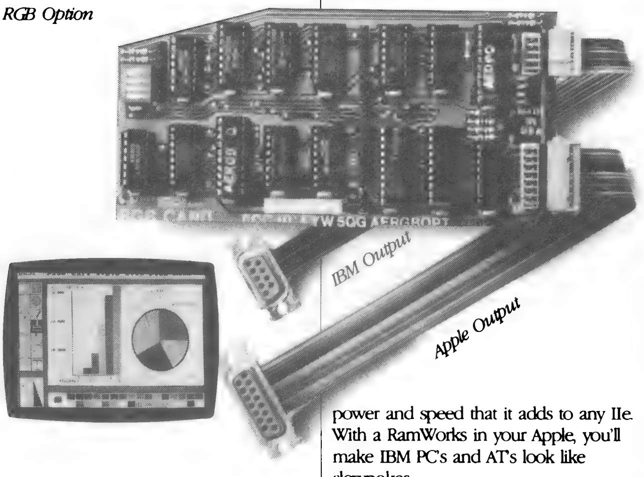
If you've got some other RAM card that's not being recognized by your programs, and you want RamWorks III, you're in luck. Because all you have to do is plug the memory chips from your current card into the expansion sockets on RamWorks to recapture most of your investment!

The Ultimate in RGB Color.

RGB color is an option on RamWorks and with good reason. Some others combine RGB color output with their memory cards, but that's unfair for those who don't need RGB *and* for those that do. Because if you don't need RGB

Applied Engineering doesn't make you buy it, but if you want RGB output you're in for a nice surprise because the RamWorks RGB option offers better color graphics plus a more readable 80 column text (that blows away any composite color monitor). For only $129 it can be added to RamWorks giving you a razor sharp, vivid brilliance that most claim is the best they have ever seen. You'll also appreciate the multiple text colors (others only have green) that come standard. But the RamWorks RGB option is more than just the ultimate in color output because unlike others, it's fully compatible with all the Apple standards for RGB output control, making it more compatible with off-the-shelf software. With its FCC certified design, you can use almost any RGB monitor because only the new RamWorks RGB option provides both Apple standard and IBM standard RGB outputs (cables included). The RGB option plugs into the back of RamWorks with no slot 1 interference and remember you can order the RGB option with your RamWorks or add it on at a later date.

RGB Option



True 65C816 16 Bit Power.

RamWorks III has a built-in 65C816 CPU port for direct connection to our optional 65C816 card. The only one capable of linearly addressing more than 1 meg of memory for power applications like running the Lotus 1-2-3™ compatible program, VIP Professional. Our 65C816 card does not use another slot but replaces the 65C02 yet maintains full 8 bit compatibility.

Endorsed by the Experts.

A+ magazine said "Applied Engineering's RamWorks is a boon to those who must use large files with AppleWorks...I like the product so much that I am buying one for my own system." inCider magazine said "RamWorks is the most powerful auxiliary slot memory card available for your IIe, and I rate it four stars...For my money, Applied Engineering's RamWorks is king of the hill."

"I wanted a memory card for my Apple that was fast, easy to use, and very compatible; so I bought RamWorks."

Steve Wozniak, the creator of Apple Computer

Apple experts everywhere are impressed by RamWorks's expandability, versatility, ease of use, and the sheer power and speed that it adds to any IIe. With a RamWorks in your Apple, you'll make IBM PC's and AT's look like slowpokes.

Quality and Support of the Industry Leader

RamWorks III is from Applied Engineering, the largest, most well supported manufacturer of Apple peripherals and the inventor of large RAM cards for the Apple. With our 5 year no hassle warranty and outstanding technical support, you're assured of the most trouble free product you can buy.

It's Got It All

- 15 day money back guarantee
- 5 year hassle free warranty insures coverage no matter where you purchase
- Built-in super sharp 80 column display, (U.S. Patent #4601081)
- Expandable to 1 MEG on main card
- Expandable to 16 meg with expander cards, with NO slot 1 interference
- Can use 64K or 256K RAMs
- Powerful linear addressing 16 bit coprocessor port
- Automatic AppleWorks expansion up to 3017K desktop
- Accelerates AppleWorks
- Built-in AppleWorks printer buffer
- The only large RAM card that's 100% compatible with all IIe software
- RamDrive™ the ultimate disk emulation software included free
- Memory is easily partitioned allowing many programs in memory at once
- Compatible, RGB option featuring ultra high resolution color graphics and multiple text colors, with cables for both Apple and IBM type monitors
- Built-in self diagnostics software
- Lowest power consumption (U.S. Patent #4601081)
- Takes only one slot (auxiliary) even when fully expanded
- Socketed and user upgradeable
- Software industry standard
- Advanced Computer Aided Design
- Used by Apple Computer, Steve Wozniak and virtually all software companies
- Displays date and time on the AppleWorks screen with any PRO-DOS compatible clock
- Much, much more!

RamWorks III with 64K	$179
RamWorks III with 256K	$219
RamWorks III with 512K	$269
RamWorks III with 1 MEG	$369
RamWorks III with 1.5 MEG	$539
RamWorks III with 2 to 16 MEG	CALL
65C816 16 Bit Card	$159
RGB Option	$129
Optional Software:	
Pinpoint with RAM Enhancement Software	$79
VIP Professional	$219

RamWorks III. The industry standard for memory expansion of the Apple IIe.

ORDER YOUR RamWorks III TODAY. 9 a.m. to 11 p.m. 7 days, or send check or money order to Applied Engineering.

MasterCard, Visa and C.O.D. welcome. Texas residents add 5⅛% sales tax. Add $10.00 if outside U.S.A.

Applied Engineering™
P.O. Box 798, Carrollton, TX 75006
(214) 241-6060

```
DE84- 78        2250        SEI             Disable IRQ's
DE85- 4E 9B BF  2260        LSR MLI.ACTIVE.FLAG  Clear this flag
DE88- 68        2270        PLA             Get saved status
DE89- AA        2280        TAX             and keep it in X-reg
DE8A- AD 9D BF  2290        LDA MLI.RETURN+1
DE8D- 48        2300        PHA             Put return address on stack
DE8E- AD 9C BF  2310        LDA MLI.RETURN
DE91- 48        2320        PHA
DE92- 8A        2330        TXA             Now push the status for RTI
DE93- 48        2340        PHA
DE94- 98        2350        TYA             Get error code in A-reg
DE95- AE 9E BF  2360        LDX MLI.X       Restore X and Y
DE98- AC 9F BF  2370        LDY MLI.Y
DE9B- 48        2380        PHA             Error code on stack
DE9C- AD F4 BF  2390        LDA E000.BYTE
DE9F- 4C A0 BF  2400        JMP LC.BRIDGE.EXIT
                2410 *--------------------------------
                2420 *   LC.BRIDGE.EXIT is code at $BFA0 in
                2430 *   the system global page.  It restores
                2440 *   the language card to the state it
                2450 *   was in when JSR $BF00 was exectuted.
                2460 *--------------------------------
                2470 *  LC.BRIDGE.EXIT EOR $E000
                2480 *                 BEQ .1   BFAA
                2490 *                 STA $C082
                2500 *                 BNE .2   BFB5
                2510 *  .1             LDA D000.BYTE  $BFF5
                2520 *                 EOR $D000
                2530 *                 BEQ .2   BFB5
                2540 *                 LDA $C083
                2550 *  .2             PLA
                2560 *                 RTI
                2570 *--------------------------------
                2580 ERR.NO.DEVICE
DEA2- A9 28     2590        LDA #$28        "NO DEVICE CONNECTED"
DEA4- 20 09 BF  2600        JSR CALL.SYSERR
                2610 ERR.CALL.NO
DEA7- A9 01     2620        LDA #1          "BAD CALL TYPE"
DEA9- D0 02     2630        BNE DEAD
                2640 ERR.PARM.CNT
DEAB- A9 04     2650        LDA #4          "BAD PARAMETER COUNT"
DEAD- 20 D7 DE  2660 DEAD   JSR CALL.CALL.SYSERR
DEB0- B0 C6     2670        BCS EXIT.TO.CALLER  ...ALWAYS
                2680 *--------------------------------
                2690 BLOCK.IO.SETUP
DEB2- A0 05     2700        LDY #5          COPY REST OF COMMAND BLOCK
DEB4- 08        2710        PHP             FROM IOB TO ZERO-PAGE
DEB5- 78        2720        SEI             DO NOT ALLOW IRQ'S
DEB6- B1 40     2730 .1     LDA (PARM.PNTR),Y
DEB8- 99 42 00  2740        STA COMMAND,Y
DEBB- 88        2750        DEY
DEBC- D0 F8     2760        BNE .1
DEBE- A6 45     2770        LDX BUFF.PNTR+1
DEC0- 86 4F     2780        STX GEN.PNTR2+1
DEC2- E8        2790        INX
DEC3- E8        2800        INX
DEC4- A5 44     2810        LDA BUFF.PNTR
DEC6- F0 01     2820        BEQ .2
DEC8- E8        2830        INX
DEC9- 20 9F FC  2840 .2     JSR CHECK.IF.MEM.FREE
DECC- B0 08     2850        BCS .3          ...NOT FREE
DECE- 20 DA DE  2860        JSR BLOCK.IO
DED1- B0 03     2870        BCS .3          ...I/O ERROR
DED3- 28        2880        PLP             RESTORE IRQ STATUS
DED4- 18        2890        CLC             NO ERRORS
DED5- 60        2900        RTS
                2910 *--------------------------------
DED6- 28        2920 .3     PLP             RESTORE IRQ STATUS
DED7- 20 09 BF  2930 CALL.CALL.SYSERR JSR CALL.SYSERR
                2940 *--------------------------------
                2950 * (DEDA) DECE EC0A EE83 F0E4 F475   callers
                2960 BLOCK.IO
DEDA- A5 43     2970        LDA UNIT.NO  Clean this up a little
DEDC- 29 F0     2980        AND #$F0
DEDE- 85 43     2990        STA UNIT.NO
DEE0- 4A        3000        LSR             Make it into index too
DEE1- 4A        3010        LSR
DEE2- 4A        3020        LSR
DEE3- AA        3030        TAX
```

```
DEE4- BD 10 BF          LDA DRIVER.ADDR.TABLE,X
DEE7- 8D F5 FE          STA JUMP
DEEA- BD 11 BF          LDA DRIVER.ADDR.TABLE+1,X
DEED- 8D F6 FE          STA JUMP+1
DEF0- 6C F5 FE          JMP (JUMP)
               *-------------------------------
                        .OR $FD65
                        .TA $800
               COMMAND.HASH.TABLE
FD65- D3 00 00
FD68- 00 40 41
FD6B- 00 00             .HS D3.00.00.00.40.41.00.00
FD6D- 80 81 82
FD70- 65 C0 C1
FD73- C2 C3             .HS 80.81.82.65.C0.C1.C2.C3
FD75- C4 C5 C6
FD78- C7 C8 C9
FD7B- CA CB             .HS C4.C5.C6.C7.C8.C9.CA.CB
FD7D- CC CD CE
FD80- CF 00 D0
FD83- D1 D2             .HS CC.CD.CE.CF.00.D0.D1.D2
               PARM.CNT.TABLE
FD85- 02 FF FF
FD88- FF 02 01
FD8B- FF FF             .HS 02.FF.FF.FF.02.01.FF.FF
FD8D- 03 03 00
FD90- 04 07 01
FD93- 02 07             .HS 03.03.00.04.07.01.02.07
FD95- 0A 02 01
FD98- 01 03 03
FD9B- 04 04             .HS 0A.02.01.01.03.03.04.04
FD9D- 01 01 02
FDA0- 02 FF 02
FDA3- 02 02             .HS 01.01.02.02.FF.02.02.02
               *-------------------------------
```

Another Update to Bob's ProDOS Program Selector
....Bob Sander-Cederlof

The following refers back to the new ProDOS Quit Code I wrote and published in the July 86 issue of AAL. It has been very popular, judging from the number of letters and phone calls we have received.

Eric Trehus (T'n'T Software) pointed out that I ignored one or more of the conventions Apple established for Quit-Code Program Selectors. On page 87 of the ProDOS Technical Reference Manual, the paragraph with number 2 states that the name of the system program should be stored in a buffer at $280, starting with a length byte. The first paragraph on page 88 says any non-standard Quit Code must begin with a CLD instruction, so programs can tell who loaded them.

If you want the CLD instruction there, go ahead and insert one between lines 1310 and 1320. I have not found it necessary for any programs I use.

Eric says that when going from BASIC.SYSTEM to APLWORKS.SYSTEM he needed the program name stroed in $280. I have never run into the problem, but it is easy to fix. Eric suggested inserting the following two lines:

```
       2065    STA $280
       2125    STA $280,Y
```

[Eric's change takes six bytes, so you need to be sure the code still fits in $300 bytes.]

If you do it Eric's way, only the name of the system file gets stored, without any prefix. I wondered whether or not a full pathname should be there, so I consulted Gary Little's "Apple ProDOS--Advanced Features" book. On page 141, near the bottom, he says either a full or a partial pathname should be put at $280. We can get the full pathname into $280 without Eric's two lines, by simply changing line 4860 from "PATHNAME .BS 64" to "PATHNAME .EQ $280". This is my preference.

When I was trying out the above, I stumbled across a problem. If my Selector finds no SYS or DIR files in a directory, it still displays the pathname and prompt messages. If you then type the RETURN key, it may try to execute garbage, or try various other things. The only valid keystroke when no files are listed is ESCAPE, which will take you back to the list of volume names. Adding two lines makes it go there without displaying the empty list:

```
       1771    TXA      see if any files listed
       1772    BEQ .2   ...none listed, start over
```

We noticed the other day that when we ran Erv Edge's correction to my program (Aug 86, page 1), we reversed the information. We said change line 3390 from BNE .1 to BPL .1; actually, it is the other way around: change from BPL to BNE. Most of you figured that out already, but we are sorry for the confusion.

Beginning to Peek at the //gs Monitor.....Bob Sander-Cederlof

The //gs Monitor has a lot of new features not found in any earlier model Apple. Unfortunately, you do not get any documentation about the monitor with your machine! Next year you will be able to buy a book that will tell you about it, but who wants to wait?

If you go into the monitor and try some of the old commands, you will find that most of them work. Memory display now shows an additional two digits of address, the bank number, and then a slash and the rest of the address. You can enter addresses the same way, so you can display any memory in the entire 16-megabyte range. For example, to disassemble ROM inside the monitor at $FDF0, type "FF/FDF0L". To look at the range of memory in bank FE from 0 to FF, type "FE/0.FF" and a <RETURN>. Note that the disassembler output looks a little different now. There are no dollar signs for hex values, and all opcodes for the 65816 are disassembled. Also note that in memory range display, you get both hex and ASCII values for each byte. If you are in 80-column mode, range display shows 16 bytes per line rather than only 8.

The new monitor preserves almost all the standard entry points, so they are clues to deciphering the rest. Looking at the routines TOSUB (FF/FFBE) and NXTITM (FF/FF73) I found the new addresses for the command character and branch tables. The command characters are in coded form at FF/F98B and following, and the branches are at FF/F9AE and following. There are 35 commands now, a fact learned by the disassembly of NXTITM.

I wrote a program to decipher the contents of these two tables and print the results. It takes a little work, because the characters in the table are not in ASCII. They correspond to ASCII values exclusive-ORed with $B0 and diminished by $89, which takes place inside the GETNUM subroutine (FF/FFA7). The addresses are the low-bytes only of entry points in page $FE of bank $FF (FF/FExx). These addresses must be incremented by one to get the real entry points, because TOSUB uses them by pushing them on the stack and doing an RTS. Anyway, the following program does all the unraveling for you, and prints 35 lines of commands in the order they appear in the tables, showing the entry points for each.

Lines 1110-1170 are an overall loop which runs 35 times, to print the 35 commands. The rest is a subroutine to print one command. By removing the stars from lines 1210-1230, you can get the output in two columns. An advantage to this is that it all fits on one screen.

```
                        1000 *Save s.mon.cmd.tbl
                        1010 *--------------------------------
     FDED-              1020 COUT   .EQ $FDED
     FDDA-              1030 PRBYTE .EQ $FDDA
     FD8E-              1040 CROUT  .EQ $FD8E
                        1050 *--------------------------------
     F98B-              1060 LTRS   .EQ $F98B    Encoded Table of Letters
     F9AE-              1070 ADR.LO .EQ $F9AE    Command starts at $FExx+1
                        1080 *--------------------------------
                           1090        .OP 65816
                           1100 *--------------------------------
                           1110 PRINT.MONITOR.COMMAND.TABLE
     000800- A0 00         1120        LDY #0
     000802- 20 0D 08      1130 .1     JSR PRINT.ONE.COMMAND
```

```
000805- C8          1140         INY
000806- C0 23       1150         CPY #35       There are 35 commands in table
000808- 90 F8       1160         BCC .1
00080A- 4C 8E FD    1170         JMP CROUT
                    1180 *-------------------------------
                    1190 PRINT.ONE.COMMAND
                    1200 *---REMOVE "***" FOR 2 COLUMNS--------------|
                    1210 ***     TYA           CHECK IF ODD OR EVEN  |
                    1220 ***     LSR                                 |
                    1230 ***     BCS .0                              |
                    1240 *-------------------------------------------|
00080D- 20 8E FD    1250         JSR CROUT
                    1260 *---TAB 10 SPACES---------------
000810- A2 0A       1270 .0      LDX #10       TAB OVER 10 SPACES
000812- A9 A0       1280         LDA #" "
000814- 20 ED FD    1290 .1      JSR COUT
000817- CA          1300         DEX
000818- 10 FA       1310         BPL .1
                    1320 *---Convert char to ASCII--------
00081A- B9 8B F9    1330         LDA LTRS,Y    Value from table
00081D- 38          1340         SEC
00081E- E9 89       1350         SBC #$89      Reverse process from GETNUM
000820- 49 B0       1360         EOR #$B0
                    1370 *---Prepare char to print--------
000822- A2 A0       1380         LDX #" "      Space before regular chars
000824- C9 A0       1390         CMP #$A0
000826- B0 04       1400         BCS .2        ...not control-char
000828- A2 DE       1410         LDX #"^"      Caret before control-chars
00082A- 09 40       1420         ORA #$40      Make control-char printable
                    1430 *---Print the char--------------
00082C- 48          1440 .2      PHA           Save char itself
00082D- 8A          1450         TXA           Print Space or Caret
00082E- 20 ED FD    1460         JSR COUT
000831- 68          1470         PLA           Print char
000832- 20 ED FD    1480         JSR COUT
                    1490 *---Print the address------------
000835- A2 03       1500         LDX #3        PRINT " $FE"
000837- BD 47 08    1510 .3      LDA FE,X
00083A- 20 ED FD    1520         JSR COUT
00083D- CA          1530         DEX
00083E- 10 F7       1540         BPL .3
000840- B9 AE F9    1550         LDA ADR.LO,Y
000843- 1A          1560         INC           Add 1 because it needs it
000844- 4C DA FD    1570         JMP PRBYTE
                    1580 *-------------------------------
000847- C5 C6 A4 A0 1590 FE      .AS -/EF$ /
```

Seeing all the commands is nice, but it would be easier to read the list if they were in alphabetical order. I modified the program a little, sorted them, and printed them in the order of their ASCII values. Lines 1110-1250 now have two loops. The first one goes through the 35 commands, and stores them into two "sorting trays". I first emptied one of the "trays", by storing zeroes in all 256 locations. Then my SORT.ONE.COMMAND subroutine stores the command ASCII code into the "tray" at the position indexed by the ASCII value itself. The address byte goes into the other "tray" at the same position.

When all 35 commands have been placed into the appropriate positions in the two trays, I run another loop to print out all the non-empty positions. There it is! Simple as can be, they are sorted almost instantaneously.

Then I tried to sort them using the same technique but in the order of the addresses. This did not work, because some addresses are used by more than one command. Only the last command using a particular address printed out. Sigh....

```
                 1000 *Save s.moncmds.sort
                 1010 *-----------------------------
FDED-            1020 COUT    .EQ $FDED
FDDA-            1030 PRBYTE  .EQ $FDDA
FD8E-            1040 CROUT   .EQ $FD8E
```

```
F98B-                LTRS   .EQ $F98B    Encoded Table of Letters
F9AE-                ADR.LO .EQ $F9AE    Command starts at $FExx+1
                     *-------------------------------
                            .OP 65816
                     *-------------------------------
                     PRINT.IIGS.MONITOR.COMMANDS.SORTED
000800- 20 1D 08            JSR CLEAR.SORTING.TRAY.A
000803- A0 00               LDY #0
000805- 20 26 08     .1     JSR SORT.ONE.COMMAND
000808- C8                  INY
000809- C0 23               CPY #35      There are 35 commands in table
00080B- 90 F8               BCC .1
                     *---Print the commands-----------
00080D- A0 00               LDY #0
00080F- B9 72 08     .2     LDA SORTING.TRAY.A,Y
000812- F0 03               BEQ .3
000814- 20 3A 08            JSR PRINT.ONE.COMMAND
000817- C8           .3     INY
000818- D0 F5               BNE .2
00081A- 4C 8E FD            JMP CROUT
                     *-------------------------------
                     CLEAR.SORTING.TRAY.A
00081D- A2 00               LDX #0
00081F- 9E 72 08     .1     STZ SORTING.TRAY.A,X
000822- E8                  INX
000823- D0 FA               BNE .1
000825- 60                  RTS
                     *-------------------------------
                     SORT.ONE.COMMAND
                     *---Convert char to ASCII--------
000826- B9 8B F9            LDA LTRS,Y   Value from table
000829- 38                  SEC
00082A- E9 89               SBC #$89     Reverse process from GETNUM
00082C- 49 B0               EOR #$B0
00082E- AA                  TAX          It is the sorting index
00082F- 9D 72 08            STA SORTING.TRAY.A,X
                     *-------------------------------
000832- B9 AE F9            LDA ADR.LO,Y
000835- 1A                  INC
000836- 9D 72 09            STA SORTING.TRAY.B,X
000839- 60                  RTS
                     *-------------------------------
                     PRINT.ONE.COMMAND
00083A- 20 8E FD            JSR CROUT
                     *---TAB 10 SPACES----------------
00083D- A2 0A               LDX #10      TAB OVER 10 SPACES
00083F- A9 A0               LDA #" "
000841- 20 ED FD     .1     JSR COUT
000844- CA                  DEX
000845- 10 FA               BPL .1
                     *---Convert char to ASCII--------
000847- B9 72 08            LDA SORTING.TRAY.A,Y
                     *---Prepare char to print--------
00084A- A2 A0               LDX #" "     Space before regular chars
00084C- C9 A0               CMP #$A0
00084E- B0 04               BCS .2       ...not control-char
000850- A2 DE               LDX #"^"     Caret before control-chars
000852- 09 40               ORA #$40     Make control-char printable
                     *---Print the char---------------
000854- 48           .2     PHA          Save char itself
000855- 8A                  TXA          Print Space or Caret
000856- 20 ED FD            JSR COUT
000859- 68                  PLA          Print char
00085A- 20 ED FD            JSR COUT
                     *---Print the address------------
00085D- A2 03               LDX #3       PRINT " $FE"
00085F- BD 6E 08     .3     LDA FE,X
000862- 20 ED FD            JSR COUT
000865- CA                  DEX
000866- 10 F7               BPL .3
000868- B9 72 09            LDA SORTING.TRAY.B,Y
00086B- 4C DA FD            JMP PRBYTE
                     *-------------------------------
00086E- C5 C6 A4 A0  FE     .AS -/EF$ /
                     *-------------------------------
000872-              SORTING.TRAY.A .BS 256
000972-              SORTING.TRAY.B .BS 256
```

Corrections to HR to DHR Converter

David Johnson just called to point out a couple of corrections to his program that converts hi-res graphic images to double hi-res. First, we somehow managed to lose a line of his code. You need to add this line:

```
     2435        LSR XLATE.MONO.AUX,X
```

Another spot to change is line 2120. The LDA #$03 should really be a LDA #$02, since 3 specifies a full color image and 2 specifies black and white. David reports that 3 has always worked OK for him, so this may not make a real difference, but the specification (ProDOS Technical Note #13) calls for a 2.

An error also crept into the text. Near the end of the fourth paragraph the article says that the AUXMEM portion of the picture is copied into main memory at $4000-5FFF. What the program really does is copy main memory from $2000-3FFF to $4000-5FFF, and then transfer the AUXMEM segment into main memory at $2000-3FFF.

Demo of Two Simple //gs Tools............Bob Sander-Cederlof

The //gs Toolbox is chock full of useful tools, and no doubt you have heard about a lot of them by now. However, all of the books and articles I have found so far just describe the tools, without showing any actual addresses or tool numbers so they can be called. Most frustrating!

If you remember the article last month about reading and writing the battery RAM in the //gs, you will remember the general way all tools are called. You must be in full-16 native mode, set up the stack in just the right way, put a tool code in the X-register, and do a JSL $E10000.

The tool code is made up of two bytes. The low-order byte is the tool set number, and the high-order byte is the tool number in that tool set. One of the missing items of information in most documentation I have seen so far is the list of tool set numbers. As near as I can figure it all out at this time, the following numbers seem to be established:

Set #	Tool
$01	Tool Locater itself
$02	Memory Manager
$03	Miscellaneous Tools
$04	Graphics Core (QuickDraw?)
$05	Desk Manager
$06	Event Manager
$07	Scheduler
$08	Sound Manager (Ensoniq stuff)
$09	Front Desk Bus
$0A	SANE (Fancy Floating Point stuff)
$0B	Integer Math
$0C	Text Tools
$0D	<<<I don't know>>>
$0E-$20	Various RAM-based tools

The first eight tools in every tools set are all the same, and do not seem to be too important to the casual user. They include boot initialization code, version and status information, reset, and so on. Even a couple of spares.

Of interest to this article, tool number $2A in set $0B will convert a two-byte value to a four-character ASCII string. For example, $12AF would be converted to the four bytes $31, $32, $41, $46. These are the ASCII values for the four hexadecimal digits of $12AF. Lines 1090-1170 in the following program use this tool. Note that all inputs and outputs for the tool are handled through the stack.

Tool $20 of tool set $0C will print out a string in ASCIIZ format. ASCIIZ is a new term to me, which I discovered this week in the Microsoft book "Advanced MS-DOS". It means a string of ASCII characters terminated by a 00 byte. After lines 1140-1170 have placed the ASCII form of the number we converted into the four bytes at HEX (line 1290), we have an ASCIIZ string starting at MSG (lines 1280-1300). Pushing the

address of MSG onto the stack and calling for tool $200C will print out the string on the screen. See lines 1180-1220.

Line 1190 may need a word of explanation. The tool needs a four-byte address on the stack, so line 1190 pushes the high-order two bytes of the address. I could have used "PEA MSG/65536", but I like "PEA MSG/256/256" better.

If you have a //gs, try out this little example. It will get you started in understanding all the new tools you have in your toolbox. If you don't have a //gs yet, start saving your nickels!

```
                  1000 *SAVE S.PRINT.DEMO
                  1010 *--------------------------------
                  1020        .OP 65816
                  1030 T
000800- 18        1040        clc          GO INTO NATIVE MODE
000801- FB        1050        xce
000802- C2 30     1060        rep #$30     FULL 16
000804- A9 34 12  1070        lda ##$1234  SAMPLE VALUE IN A-REG
                  1080 *---Tool: convert hex to string-------
000807- F4 00 00  1090        PEA 0        4-BYTES FOR RESULT
00080A- F4 00 00  1100        PEA 0
00080D- 48        1110        PHA          VALUE TO BE CONVERTED
00080E- A2 0B 2A  1120        LDX ##$2A0B  TOOL: HEX-VALUE TO ASCII
000811- 22 00 00 E1 1130      JSL $E10000
000815- 68        1140        PLA          TWO CHARS
000816- 8D 4E 08  1150        STA HEX      INTO STRING
000819- 68        1160        PLA          TWO MORE CHARS
00081A- 8D 50 08  1170        STA HEX+2    INTO STRING
                  1180 *---Tool: print a string--------------
00081D- F4 00 00  1190        PEA MSG/256/256
000820- F4 2D 08  1200        PEA MSG      ADDRESS OF STRING
000823- A2 0C 20  1210        LDX ##$200C  TOOL: PRINT STRING
000826- 22 00 00 E1 1220      JSL $E10000
                  1230 *--------------------------------
00082A- 38        1240        SEC          RETURN IN EMULATION MODE
00082B- FB        1250        XCE
00082C- 60        1260        RTS
                  1270 *--------------------------------
00082D- 54 48 45 20
000831- 4E 55 4D 42
000835- 45 52 20 49
000839- 4E 20 54 48
00083D- 45 20 41 2D
000841- 52 45 47 49
000845- 53 54 45 52
000849- 20 49 53 20
00084D- 24        1280 MSG    .AS "THE NUMBER IN THE A-REGISTER IS $"
00084E-           1290 HEX    .BS 4
000852- 0D 00     1300        .HS 0D.00
                  1310 *--------------------------------
```

Woz Re-Codes Hi-Res Address Calculations....Bob Sander-Cederlof

In the October or November issue of the Washington Apple Pi newsletter, Rick Chapman wrote a review of various methods of calculating the hi-res base addresses. Steve Wozniak liked the article, and responded with a long "letter to the editor" in the December issue. Steve also presented a new version of the hi-res address calculator which is both shorter and faster. In fact, as far as I am aware, it is the fastest method ever, except for table-lookups.

In the September 1983 issue of Apple Assembly Line, I presented both the original Woz code and a shorter-faster version by Harry Cheung of Nigeria. Here are the specs:

```
       Applesoft ROM version:  33 bytes, 61 cycles
        Harry Cheung version:  25 bytes, 46 cycles
         New Wozniak version:  26 bytes, 36 or 37 cycles
```

The byte counts do not include an RTS at the end of the code, nor do the times include a JSR-RTS. After all, if you are really working for speed you will put the code in its place, not make it a subroutine.

Woz's new version takes either 36 or 37 cycles, depending on the values for the first two bits of the line number. Remember that the line number can be any value from 0 to 191, or $00...$BF. That means the first two bits are either 00, 01, or 10. If you look at lines 1090-1120 below, you will see that the shortest path is for 00, taking both branches, giving a running time for the whole calculation of 36 cycles. If the first two bits are 01 or 10, one branch will be taken and the other not, making the total time 37 cycles. In Woz's letter he shortchanged himself, thinking possibly both branches might not be taken, giving a total running time of 38 cycles; this cannot happen with legal line numbers.

Line 1180 adds in either $10 or $20, depending on which hi-res page you are using. The Applesoft code here adds in a value of either $20 or $40, so if this version were to be inserted into Applesoft the generation of HPAG2 would have to be changed. No problem, and not likely anyway. By the way, if you are only using one specific hi-res page, you can change line 1180 to an immediate mode form, saving yet another cycle.

Here is Woz's new version, reformatted for the S-C Assembler and with some changes in comments:

```
                  1000 *SAVE NEW.WOZ.HIRES.CALC
                  1010 *--------------------------------
26-               1020 GBASL   .EQ $26
27-               1030 GBASH   .EQ $27
E6-               1040 HPAG2   .EQ $E6       Applesoft puts it here anyway.
                  1050 *--------------------------------
0800- 0A          1060 CALC    ASL           A--BCDEFGH0
0801- AA          1070         TAX           TAX...TXA could be TAY...TYA
0802- 29 F0       1080         AND #$F0      A--BCDE0000
0804- 10 02       1090         BPL .1        B=0
0806- 09 05       1100         ORA #$05      A--BCDE0B0B
0808- 90 02       1110 .1      BCC .2        A=0
080A- 09 0A       1120         ORA #$0A      A--BCDEABAB
080C- 0A          1130 .2      ASL           B--CDEABAB0
080D- 0A          1140         ASL           C--DEABAB00
080E- 85 26       1150         STA GBASL
0810- 8A          1160         TXA           C--BCDEFGH0
0811- 29 0E       1170         AND #$0E      C--0000FGH0
0813- 65 E6       1180         ADC HPAG2     0--00xxFGHC
                  1190 *       HPAG2 = $10 for base $2000, $20 for base $4000
0815- 06 26       1200         ASL GBASL     D--00xxFGHC  GBASL=EABAB000
0817- 2A          1210         ROL           0--0xxFGHCD
0818- 85 27       1220         STA GBASH
081A- 60          1230         RTS
                  1240 *--------------------------------
```

Apple Assembly Line (ISSN 0889-4302) is published monthly by S-C SOFTWARE CORPORATION, P.O. Box 280300, Dallas, Texas 75228. Phone (214) 324-2050. Subscription rate is $18 per year in the USA, sent Bulk Mail; add $3 for First Class postage in USA, Canada, and Mexico; add $14 postage for other countries. Back issues are available for $1.80 each (other countries add $1 per back issue for postage). A subscription to the newsletter and the Monthly Disk containing all source code is $64 per year in the US, Canada and Mexico, and $87 to other countries.